Adaptive Behavioral Programming

Nir Eitan and David Harel
Dept. of Computer Science and Applied Mathematics
Weizmann Institute of Science, Israel
firstname.lastname@weizmann.ac.il

***Abstract*—We introduce a way to program adaptive reactive systems, using behavioral, scenario-based programming. Extending the semantics of live sequence charts with reinforcements allows the programmer not only to specify what the system should do or must not do, but also what it should try to do, in an intuitive and incremental way. By integrating scenario-based programs with reinforcement learning methods, the program can adapt to the environment, and try to achieve the desired goals. Visualization methods and modular learning decompositions, based on the unique structure of the program, are suggested, and result in an efficient development process and a fast learning rate.**

***Keywords*-scenario-based programming; adaptive systems; reinforcement learning; behavior-based; LSC ; BPJ**

I. Introduction

The design and development of reactive systems, whose role is to maintain an ongoing interaction with their environment, is a research area with many challenges, and in recent years has been the subject of considerable work. In 1999, an extension of the language of *message sequence charts* (MSCs), termed *live sequence charts* (LSCs) was introduced by Damm and Harel [1], in order to describe the inter-object based behavior of reactive systems.

The LSCs language embodies the scenario-based programming approach, whose main goal and dream is to make programming more natural, and to render the process a smooth extension of the way we think [2]. Lately, we have begun to call the general paradigm underlying executable scenarios *behavioral programming*; see [3], [4]. Scenarios in LSC contain multi-modal actions and conditions, and can describe what should be done, what can be done and what may not be done. These scenarios are programmed independently, or semi-independently, in a modular fashion, and are later interleaved to dynamically obey the rules they state and thus form an executable program; see [5].

In this paper we propose to give the user the possibility to not only set these multi-modal rules and boundaries, but also to specify a desired behavior in an easy and intuitive way, by setting the goals the program should achieve, and the scenarios it should avoid. The program is then given the freedom to learn and adapt according to its experience from the environment. Such adaptivity is highly desirable, as the programmer cannot always anticipate how the reactive system's environment will react. In addition, it allows the programmer to focus on specifying his/her needs, while leaving the details and optimizations to the program, thus emphasizing the advantages of behavioral programming, and offering a natural and modular way to program.

Specifying desired behavior is done by associating scenarios with rewards and punishments. As the model is executed, it learns the best behavior according to these reinforcements: for every state, represented by the progress cuts in the LSCs, it learns from experience what the preferred action is for achieving optimized expected reward. This is done by using reinforcement learning methods and integrating the resulting policy into the action selection mechanism, which is used whenever multiple events are available.

Unfortunately, the standard way of doing this gives rise to an exponentially growing number of possible states. In many cases, optimizing is a necessity for obtaining a feasible learning process, especially in dynamic environments. We suggest three learning methods that exploit the unique structure of the scenario-based program, yet retain the modularity and simplicity of the programming process. The performance of these methods is compared for two sample programs.

We also address visualization issues. Analyzing program execution traces and being able to understand the resulting sequences of actions are critical stages in the development process, and can be helpful in detecting bugs and updating the program's goals.

II. Integrating Reinforcements into Behavioral Programs

A. Implementing scenario-based programming

The scenario-based approach to behavioral programming can be carried out in various ways. The *Play-Engine* tool was built in 2003 to support LSCs and the *play-in* and *play-out* approaches for programming and executing them, respectively [5]. More recently, the first version of a new, more powerful tool, *PlayGo*, has been completed [6]. Both tools allow system designers to play-in the required multi-modal scenarios of behavior via a user-defined GUI, and the scenarios are translated on the fly into LSCs. The resulting LSC specification can then be fully played out (executed), using a coordinated means for choosing an appropriate event for execution at each stage, which adheres to the multi-modal nature of the state of each chart. All the events in

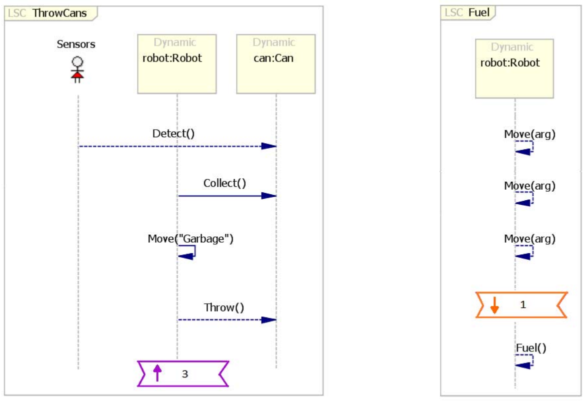


Figure 1. Two goal LSCs for a can-picking robot. The `ThrowCans` LSC describes a scenario in which whenever a can is detected, the robot can collect it, move to the garbage, and dispose of it. Whenever this is accomplished, a reinforcement of $+3$ is given. The `Fuel` LSC describes a scenario in which whenever three moves are carried out without charging in between, a small negative reinforcement, of -1, is given.

the specification that are unifiable with the chosen event are advanced too; see [5].

A Java-based counterpart to LSCs has also been developed recently, via the *BPJ* library (for behavioral programing in Java) [3]. Scenarios are programmed using Java *b-threads*, which are synchronized periodically at given syncpoints. At each syncpoint, a b-thread can be set to request, wait for and block (i.e., forbid) specific events. Similarly to the LSC play-out process, at each step during execution, a centralized arbitrator chooses an event that is enabled for execution, meaning that it is requested but not blocked. The event is executed, and all scenario instances waiting for that event are advanced.

In this paper, we suggest ways to modify these choice and execution mechanisms, which, coupled with appropriate extensions to the LSC language and BPJ, render both languages suitable for dealing with adaptive behavior.

B. Extending the LSC language

We suggest to extend the syntax of LSCs with the notion of reinforcements, in order to describe desired and non-desired behaviors. The syntax is similar to the way assignments appear in the language, with the ability to visually bind (i.e., sync) the reinforcement signal to one or more of the LSC lifelines. Whenever the LSC advances and all the synced lifelines reach the reinforcement position, the reinforcement is enabled and given to the program, teaching it to either avoid or try to repeat the executed behavior. A numerical value, positive or negative, is assigned to the reinforcement, indicating how desirable reaching this reinforcement is.

Figure 1 shows two almost self-explanatory LSCs that demonstrate adding positive and negative reinforcements.

C. Modeling LSCs as a Markov decision process

To use reinforcement learning methods, we first model the scenario-based program as a Markov decision process (MDP), a widely-used approach to sequential stochastic decision problems. An MDP is a 4-tuple (S, A, P, R), where S is a finite state space; $A(s)$ is a finite action space given the state s; $P_a(s, s')$ corresponds to the probability of moving from state s to state s' by executing action a; and $R_a(s, s')$ is the reward given when moving from state s to s' by executing the action a. A reinforcement learning task that satisfies the Markov property — i.e., a memory-less process such that the current state provides the best possible basis for choosing an action — can be modeled as an MDP [7]. The optimal policy, that is a mapping between states and possible actions that maximizes the expected utility, can then be found; at least in principle.

Let the scenario-state at a given time be the cut of the corresponding LSC, augmented by the current valuation of internal variables. The states of the MDP for an LSC program correspond to the Cartesian product of all its scenario-states, $S = \prod_{l \in LSCs} S_l$. The action space A_s is set to consist of all enabled events in state s, and $R_a(s, s')$ is the sum of reinforcements given in all the charts that advanced after the event a was executed.

After an action is selected and executed, an external event, such as a user action or the completion of a sensor reading, may occur. Only after all external events are executed, and the system can choose an internal event to execute, does the system reach a new Markov state, and the Markov transition ends. The transition probability P of moving from state s to s' given action a is therefore dictated by the external events of the system. Typically, this probability function will not be foreseen when programming the system, and must be learned from experience, using reinforcement learning methods.

We note that a single LSC can seldom be modeled as an MDP since, given a scenario state, the set of possible actions is not even fixed: another scenario can block the execution of one or more of its actions, or might cause another event request. In addition, for many kinds of systems with an unknown environment, programming the system with pure Markov states, where the environment behavior depends only on the system's state, may be hard or impossible. However, given a detailed enough state-space, an approximation to a Markov state can be obtained, which is generally not considered to be a severe problem for a reinforcement learning agent [7]. The more detailed the program is, describing more possible scenarios that may affect the environment, the better the optimal policy will be, though it will usually take longer to be learned.

D. Scenario classification

Different scenarios may have different functions in the learning mechanism. Some may set goals and prescribe behavior, for others the goal is negative — i.e., to forbid certain behavior — and some may not participate in the event selection mechanism at all. The programmer can set the learning functionality of the scenarios according to the following classification:

- Goal-Scenarios: ones that grant reinforcements.
- Base-Scenarios: ones that may affect the run or increase the state-space, but do not give any reinforcements.
- Auxiliary-Scenarios: ones that do not participate in the learning at all, and are used to monitor the execution, as test-cases, or for user interface functions.

This classification is demonstrated in section V.

III. Learning and Adaptivity

A. Temporal difference learning

The underlying idea of reinforcement learning is to learn what to do in order to maximize some notion of cumulative reward. The program is not told what actions to take for every situation, but rather discovers this by trying [7], receiving rewards or punishments in the process.

In the current work, the reinforcement learning problem is solved using *temporal difference* (TD) learning [7]. TD learning works directly from raw experience without having to know the exact environment model (thus, it is model-free), but keeps updating its estimate as it progresses. This online characteristic of the learning makes it extremely suitable for reactive systems that run indefinitely, or for very lengthy periods. The model-free feature makes it more adaptable and easier for the programmer, who doesn't always know all the parameters of the system, or can predict the environment.

For every state, and for every possible action given that state, an expected reward prediction $Q(s, a)$ is retained, and is dynamically updated according to the actual reinforcements given. We use the SARSA algorithm for this, which has been shown to be highly beneficial in modular learning [8], [9]. The SARSA algorithm is on-policy, so $Q(s, a)$ is updated according to the used policy and the actual next action taken, and it allows convergence of the Q-values to the optimal ones [10] (under certain conditions). A tradeoff between exploration, the ability to look for a new strategy, and exploitation, using what the agent already knows about the environment, can be made by using the softmax action selection rules [7].

B. Modular scenario-based learning

Basic reinforcement learning algorithms do not scale up easily; their performance tends to degrade rapidly as the size of the state space increases. Generalization and function approximation methods that make it possible to use a limited subset of the state space have been extensively studied, and various solutions are being researched and used — neural networks and linear approximation, for example. Most of these methods, however, require the programmer to tailor a specialized solution to a given problem, whereas our quest is to seek easy-to-use and generalized methods.

Our suggestion for generalized optimizations is based on the *modular learning* approach, as suggested in [11], where the program is decomposed into components (sub-agents), each using its own predefined state space. The program's decomposition and the state space of each component are usually assumed to be known and implemented by the user. We propose optimizations for use in behavioral, scenario-based programming, for which no further work is required, thus easing the programming process itself and retaining the benefits of scenario-based programs, such as behavioral modularity. While the solutions afforded by our method might be further from the optimal than those of some of the specialized methods, we believe that the simplicity in programming and maintenance makes it beneficial for adaptive system programming.

To implement modular learning in the scenario-based paradigm, a centralized arbitrator first derives the enabled actions. Then, the goal-scenarios of the program, functioning as the modular learning sub-agents, place their votes for every enabled action by sending the arbitrator their corresponding Q-values. The arbitrator then selects an action according to the sum of votes $\sum_j Q_j(s, a)$, thus maximizing the average satisfaction of the goal-scenarios in taking an action; see [8] for an overview of arbitration functions. Each such sub-agent maintains its own Q-value table, and learns only according to its own reinforcements.

The problem of determining the state space of each of the sub-agents is yet to be solved. We define S_u to be the scenario-state of scenario u, and $\widetilde{S_u}$ the overall state-space of that scenario; i.e., the states it can perceive. V is the set of all the scenarios of the program, excluding auxiliary-ones.

The naive approach is to let each goal-scenario learn and make decisions according to the overall program state space, which is the Cartesian product of all the states of all active non-auxiliary scenarios, $\widetilde{S_u} = \prod_{v \in V} S_v$. We call this method *full learning*. Using the on-policy reinforcement learning algorithm SARSA on each sub-agent separately, a globally optimal policy is achieved [9].

This optimality is only assured as long as each sub-agent updates its Q-table according to the overall program state. Unfortunately, using such a globally shared state space may result in slow learning, with intensive memory and CPU requirements, as the number of states and size of the Q-table can be exponential in the number of scenarios.

In the *egocentric scenario learning* method, on the other hand, we allow no collaboration between scenarios. Each tries to push for its own goal without knowing the state of the others, and its perceptual horizon is only set on itself, $\widetilde{S_u} = S_u$. The main benefit of this method is the greatly reduced state space, which is now polynomial in the number of scenarios, making it feasible to run large-scale reactive programs. In addition, adding a scenario does not affect the state spaces of existing scenarios, resulting in better modularity. On the other hand, the learned policy may be highly suboptimal.

The third scenario-based learning method we suggest is *perceptual scenario learning*. It makes a compromise in state-space size between full learning and the egocentric method. The perceptual method is based on self-learning the inter-dependency graph of the scenarios during program execution, and using it to set the perceptual horizon of each scenario. Each scenario then learns only according to the scenarios that can affect it.

We define the inter-dependency graph as follows. Its vertex set V represents all scenarios, excluding auxiliary ones. A directed edge exists from scenario u to scenario v iff the state of v directly affects the state transition of scenario u; i.e., iff v blocks u's requested events, thus disabling an event transition, or v requests something that u is waiting for, thus enabling a transition, or v interrupts u, ending the scenario.

We say that scenario u *depends on* scenario v iff there is a path from u to v in the inter-dependency graph. Let $D(u)$ be the set of scenarios upon which u depends. The state-space of a goal-scenario u is then set to be the Cartesian product of all states upon which u depends, $\widetilde{S_u} = \prod_{v \in D(u)} S_v$.

To prevent overestimation of the dependencies, we use a dynamically-built inter-dependency graph, constructed on the fly during execution. It counts dependencies that actually occur during the execution, and ignores states that are unreachable. When an edge is added, the Q-value is updated, $Q(s_{new} \times \prod_{i=1}^{m} s_i, a) \leftarrow Q(\prod_{i=1}^{m} s_i, a)$, reusing the previously acquired knowledge.

Although each sub-agent can be modeled as an MDP (given a state, the action set is now fixed), there is still no guarantee of convergence to the optimal policy, since the sub-agents still interact with each other, via the central arbitrator and the action selection mechanism. Examples that do not converge to the optimal solution can be easily given. However, it has been shown that even though perceptual aliasing might prevent convergence to the optimal strategy, the on-policy SARSA algorithm usually yields good results [8], [12], given a good decomposition.

C. Coordinating the reinforcements

While the scenario-based approach allows modularity in setting the goals, setting the reinforcement of a newly-added goal in a way that is consistent with the previously programmed scenarios might be tricky. In some cases, the reinforcements might not be comparable between the scenarios, and no ideal arbitrator is possible [13].

To partially deal with this problem and retain programming modularity, a ground scale was given to the reinforcements, by using reinforcement constants. Instead of setting a numerical reinforcement, the programmer can set the reinforcement to be, for example, REWARDS.HIGH, in this case indicating a highly desirable goal. By using consistent values, setting the reinforcements of a new scenario can be done independently of the earlier scenarios.

To further aid in balancing reinforcements of goal-scenarios, we allow the programmer to externally strengthen or weaken scenario reinforcements. In addition, static priorities may be given to scenarios, allowing the creation of layers of scenarios with a given priority. The arbitrator then selects one of the available actions that was requested by a scenario with the top priority.

D. Setting the meta-parameters

Using the SARSA algorithm, multiple meta-parameters must be set. Using such parameters wisely can greatly enhance the learning performance, but setting them properly is not an easy or intuitive task for the programmer. In the current version of our tool they are kept fixed, thus (at least for now) sacrificing optimality for user experience. The parameters were set according to their performance as tested in multiple medium-sized applications. It is assumed that the programmer uses the same reinforcement scale as used by the reinforcement constants.

In updating the Q-values, two parameters must be set: γ, the discount factor, and α, the learning rate. They determine the importance of future rewards and the importance of newly required information, respectively. In the current tool we set these as $\gamma = 0.9$, $\alpha = 0.5$. A Boltzmann distribution, with a temperature of $\tau = 3.5$, was used in the softmax action selection, to allow exploration.

Eligibility traces allow the rewards to be propagated to a sequence of the past actions. It allows faster learning [7], with some computational cost, something that seems especially important in reactive systems with a dynamic environment. It can also help in the case of perceptual aliasing [14], which can mostly happen in the egocentric or perceptual scenarios learning. A replacing eligibility trace, with $\lambda = 0.95$, was used.

IV. Visualizing Learning

A. Trace visualization

We extend previous work on a visualization tool for scenario-based programming [15], [16], which allows viewing the trace during execution and the interaction between the behavioral modules. We offer browsing, filtering, and grouping mechanisms for comprehending traces, and the following notations are added to aid in the comprehension of adaptive programs (see Figure 2):

- Reinforcements: A purple plus icon and an orange minus icon represent positive and negative reinforcements, respectively.
- Scenario-states: Locations in which a scenario changed its scenario-state are marked with a gray icon with the name of the new state.
- Program state: Colored tables, located above the names of the events that were executed at each syncpoint,

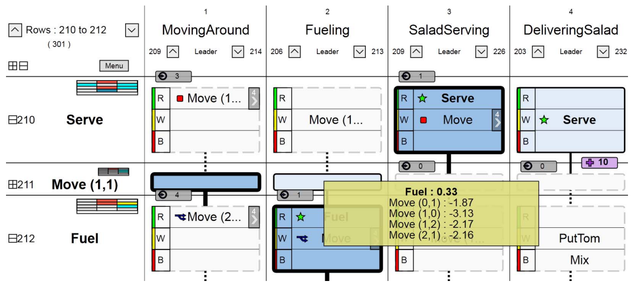


Figure 2. A partial trace visualization of the salad-cutting robot program, described in section V-A. By looking at the 212th syncpoint, one can see why the Fuel event was chosen, by comparing the total votes it was given to the alternatives, and checking why other events were not enabled.

indicate the overall program state at the syncpoint, thus enabling visually following the state changes of the program. Each table cell is colored according to a specific scenario-state.

- Paths not taken: Events that were enabled at a given syncpoint but were not executed due to the learning mechanism, are marked by a split arrow icon. Hovering over the icon, the sum of Q-values for all enabled events is shown, helping the programmer understand why a given event was executed and another was not.

B. *Inter-dependency Visualization*

The dynamic inter-dependency graph is shown during program execution, allowing the programmer to comprehend the structure of the program and the perceptual horizon of each scenario. The graph is depicted using a Java software library for data visualization, called JUNG [17], where nodes correspond to encountered scenarios and directed edges represent dependencies between scenarios. The graph allows the programmer to comprehend the structure of the program, and understand better how the scenario-based optimization works for his program.

For each goal-scenario node, the total reinforcement given so far by that scenario is displayed and is color-coded: white means the total reinforcements given by this scenario sum to 0; orange represents an "anti-goal-scenario"; that is, a scenario with total negative reward; and purple indicates a total positive reward. The stronger the color's hue, the stronger the reinforcements. Base-scenarios are colored gray and give no reinforcements. Auxiliary-scenarios are not displayed at all. See Figure 3.

V. Usage and Evaluation

We now present two examples of behavioral, scenario-based programs, and demonstrate the effectiveness of our learning methods. We consider both static and dynamic environments. The examples were implemented and tested on BPJ, and are available on [18].

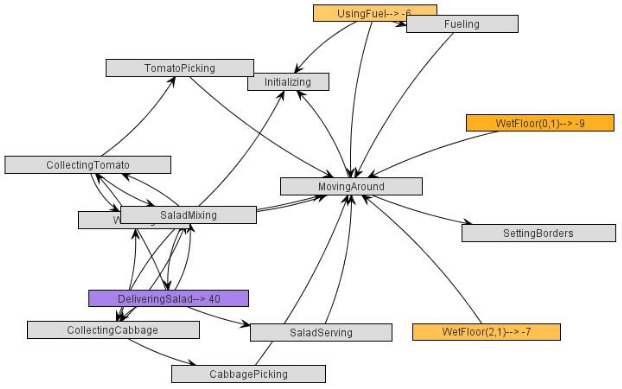


Figure 3. The inter-dependency graph of an execution of the salad-cutting robot, described in section V-A, where two of the tiles were set to be wet. The `DeliveringSalad` scenario depends, among others, on `CollectingTomato`, `SaladWashing` and `SimpleEngine`, but is independent of the fuel status and wet floors. The `UsingFuel` goal-scenario depends only on the `Fueling` scenario and the robot position (via `SimpleEngine`), and the `WetFloor` scenarios only depend on the position.

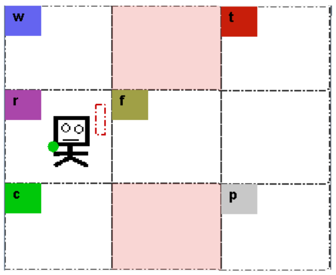


Figure 4. The salad-cutting robot GUI. The robot needs to pick a tomato from square t and a cabbage from c, wash them in w, and put them one by one in p. After mixing them together in square p, they can be served in r (the robot's current location). Fueling at square f fills the robot's fuel tank (displayed by its side), preventing low-fuel penalties. A wet floor square (in red), which can be set by the user, gives the robot an additional penalty for passing through.

A. *A salad-making robot*

In this example, the program simulates a simple robot situated on a 3x3 grid, and who needs to accomplish certain goals (see Figure 4). A `DeliveringSalad` b-thread describes a goal-scenario where after both a tomato and a cabbage are retrieved, the robot should mix them together, and then receives a very high reinforcement (+10) for serving the resulting dish. `UsingFuel` describes a scenario in which whenever the fuel is down a negative reinforcement (of -1) is given for every move, since the robot needs to use an alternate power source. Finally, `FloorWetting` specifies that from whenever a tile becomes wet until it dries, the robot receives a medium negative reinforcement (-1) for passing on it.

Base-scenarios further add details to the salad serving world, and to what the robot can do. `CollectingTomato` and `CollectingCabbage` describe the scenarios of collecting the vegetables, from picking them up, via washing them, to putting them in the kitchen; `TomatoPicking` and `CabbagePicking` describe where tomatoes and cabbages can be picked; `Washing` allows the robot to wash the vegetables; `MovingAround` describes the robot's movement (one square at a time, horizontally or vertically); `SettingBorders` set the borders of the board; and so on.

Auxiliary scenarios, which do not participate in the learning, can be added too. Here, `CoordinatingTime` and `AutomatingWetFloor` were added for debugging and testing issues, and are responsible for slowing down the simulation and simulating an environment change of floor wetting, respectively.

The results of running this example, as shown in Figure 5, point clearly to the advantage of using perceptual learning in such an application. Egocentric learning only gave the best results in the first 1000 or so moves, and learns how to save fuel quickly. However, it does not succeed in learning the salad delivery goal properly, and provides a low reinforcement rate, even after a long period of learning.

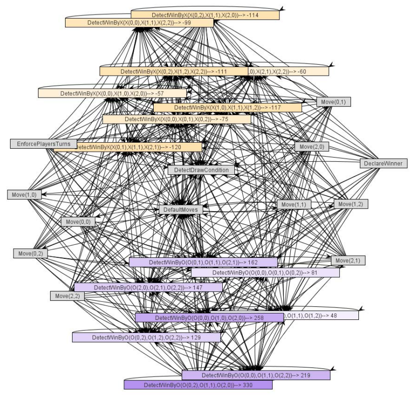
The full learning method slowly learns the task, but accumulates a high negative reward while doing so. Not only does it take it the longest to learn a good policy, but it also suffers the most from the changes in the environment (Figure 5(c)), since it remembers and keeps more information, some of it irrelevant to the achievement of specific tasks. The perceptual learner, however, learned the best policy, and did it quickly, rendering it the best solution for this example. Each sub-agent only learned according to the most relevant information, as seen in the inter-dependency graph (Figure 3), and only the position of the robot was shared between the sub-agent's state spaces.

We note that the graphs only count the number of moves it takes to achieve a good policy. The CPU time and memory that were used by the full learning method were several times larger than those used by perceptual learning, and much greater than those used by the egocentric method.

B. Board games

In this section we emphasize online learning of adaptive games, where the learner must keep its efficiency, while both the environment and the game rules might change. This view is a bit extreme for normal board games, in which the rules are fixed and a long training session is available. Nevertheless, we believe that an adaptive gaming approach is interesting, can represent a simplified model of more complex real-life problems, and may greatly benefit from the scenario-based modularity. A survey on board games and the usage of reinforcement learning to obtain a good strategy is given in [19].

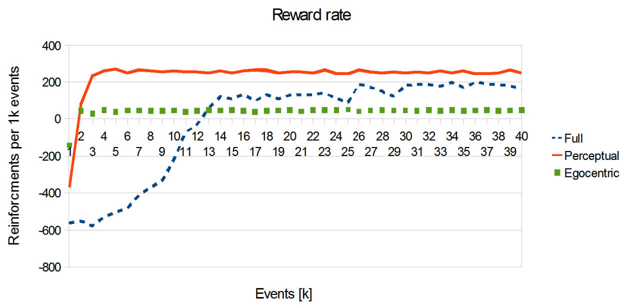
We concentrate on the classical game of Tic-Tac-Toe (TTT), where two players, X and O, take turns in marking a 3x3 board, each trying to form a horizontal, vertical or diagonal line to win. Tic-Tac-Toe is simple and has a relatively small state space. In previous work of our group TTT was programmed very naturally using the scenario-based approach [3]. Rules were given more or less independently, and they are interleaved during execution: b-threads of classes `DetectWinByX` and `DetectWinByO` detect victory conditions; `EnforcePlayersTurns` force players to alternate turns; `DisallowSquareReuse` prevents a square from being chosen twice; and `DefaultMoves` sets all the possible game moves.


DetectWinByX(X(0,2),X(1,1),X(2,0))--> -114
DetectWinByX(X(0,0),X(1,1),X(2,2))--> -99
DetectWinByX(X(0,2),X(1,2),X(2,2))--> -111
,0),X(2,1),X(2,2))--> -60
DetectWinByX(X(0,0),X(1,0),X(2,0))--> -57
Move(0,1)
DetectWinByX(X(1,0),X(1,1),X(1,2))--> -117
DetectWinByX(X(0,0),X(0,1),X(0,2))--> -75
EnforcePlayersTurns
DetectWinByX(X(0,1),X(1,1),X(2,1))--> -120
Move(2,0)
DeclareWinner
DetectDrawCondition
Move(1,0)
Move(1,1)
Move(1,2)
DefaultMoves
Move(0,0)
Move(0,2)
DetectWinByO(O(0,1),O(1,1),O(2,1))--> 162
Move(2,1)
DetectWinByO(O(0,0),O(0,1),O(0,2))--> 81
DetectWinByO(O(2,0),O(2,1),O(2,2))--> 147
Move(2,2)
DetectWinByO(O(0,0),O(1,0),O(2,0))--> 258
),O(1,1),O(1,2))--> 48
DetectWinByO(O(0,2),O(1,2),O(2,2))--> 129
DetectWinByO(O(0,0),O(1,1),O(2,2))--> 219
DetectWinByO(O(0,2),O(1,1),O(2,0))--> 330
Figure 6. The inter-dependency graph of a Tic-Tac-Toe game. Each scenario (graph node) is dependent on all others.

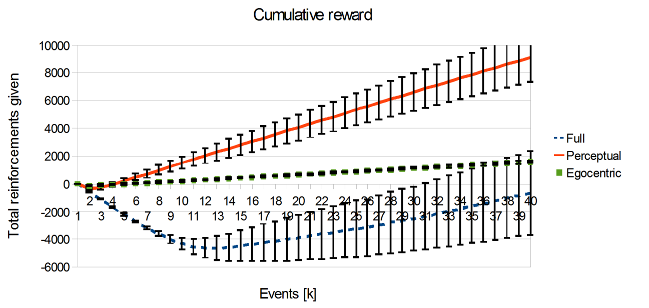
Strategy scenarios, such as scenarios that prevent the completion of an opponent's line or countering forks, can be added as well, resulting in an expert Tic-Tac-Toe player. In contrast, here we do not implement any such strategies; we use only reinforcement learning methods to learn the strategy. This allows the computer player to adapt and exploit the specific strategy its opponent is using. It also indicates how we might use this learning-program method to deal with more complex games, where the strategy is unknown, or is hard to program. Nevertheless, combining strategy scenarios that represent prior knowledge with the system learning abilities, will probably give the best results.

The inter-dependency graph of a TTT game, shown in Figure 6, has a clique between the goal-scenarios: each one depends on all the rest. The reason is the way each scenario can interrupt others: given that a line is formed and one of the goal-scenarios is accomplished, that scenario interrupts all others, prevents them from being carried out, and the game thus ends. This is the general rule in all games where only one goal can be accomplished, which indicates that the space-state of such games can probably not be easily decomposed, and perceptual learning is really the same as full learning.

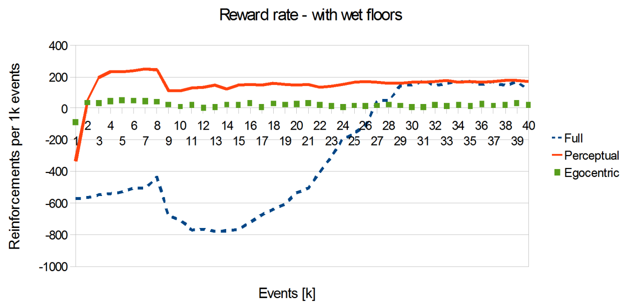
Running a sequence of games between an egocentric learner and a full learner results in a short-term advantage to the egocentric. However, this advantage is quickly replaced by the superiority of the full learner, which derives from its



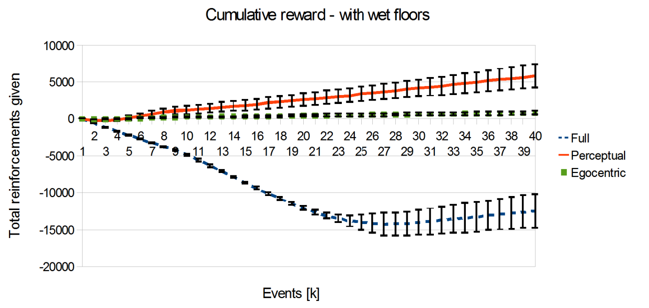
(a)



(b)



(c)



(d)

Figure 5. Results of running full learning, perceptual learning and egocentric learning on the salad-cutting robot. (a) and (c) describes the reward rate (reinforcements per 1000 games), (b) and (d) describes the cumulative reward, with its standard deviation. (a) and (b) are the results for static environment, and (c) and (d) for a dynamic environment, where a wet floor was added at tile $(0, 1)$ on the 8000th event, and another one at $(2, 1)$ on the 10,000th event. The results were averaged over 10 test episodes for each learning method

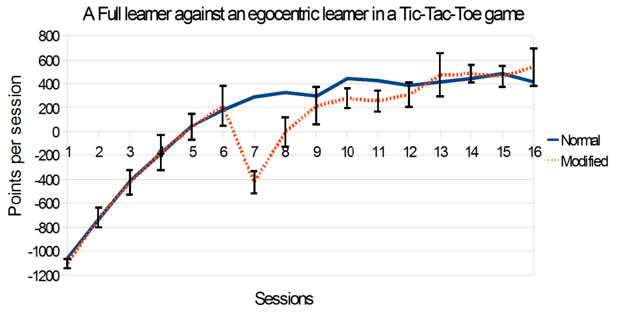


Figure 7. This figure compares the performance of the full learner against an egocentric learner, on both a normal TTT game and a modified game in which the two scenarios `3Corners` and `3Sides` were added in the 7th session. The graph shows the number of points (victories and losses) won by the full learner per session, where each session contains 2000 alternating games. The results were averaged over ten trials.

better optimal policy (see Figure 7). When the additional victory scenarios `3Corners` and `3Sides` are inserted, allowing the O player to gain two victory points for having marked either three of the corners or three of the tiles adjacent to the central one, the egocentric learner is faster to learn and take advantage of these.

VI. Related Work

The bottom-up approach that calls for modeling intelligence as simple behavioral modules and then "setting them off" to yield emergent behavior, is also part of the *behavior-based artificial intelligence* movement in AI, following Brook's *subsumption architecture* [20]. While at first no central arbitration or learning function was used, these were added in later work [21], [22]. Our own work here combines the ideas of behavior-based AI architecture with a natural and intuitive programming language, aspiring to give rise to a better programming experience.

Integrating modular reinforcement learning with a programming language is the approach taken also in [12]. The agent there consists of a set of behaviors programmed in *ABL* [23], each of which can vote for the next action to be taken. We share the goals described in that work, and believe we offer a different (and hopefully somewhat richer) kind of interaction between behaviors, and better modularity. This is because the programmer need not be concerned with agent decomposition and defining the state spaces.

VII. Conclusions and Future Work

We have suggested a way to program adaptive systems. Building upon the modularity and incrementality of the scenario-based approach, the result appears to be a natural programming experience, yielding programs that are also easier to comprehend and maintain. Setting and removing goals can be easily done, and the changes can be monitored and debugged using the tool visualization.

We have examined three learning mechanisms, all based on the scenario-based structure of the program and requiring no further work from the programmer. The egocentric method, where each scenario only cares about its own state, seems to give best results in the short term, and full learning grants a slow and promised learning rate. Perceptual learning, where each scenario only takes scenarios that affect it into account, seems to be most suitable for dynamic environments. It appears to exhibit reasonable learning time and reasonable memory and CPU requirements in many applications. However, in some cases it can be easily shown to be no better than the full learner. Clearly, further evaluation of these is still required, especially on more complex real world applications. It would also be interesting to examine properties of the inter-dependency graph on different types of adaptive systems.

Future research directions also include integrating with other work on behavioral programming. Combined with *smart play-out* [24], for example, the program could strive to reach goals while avoiding chart violations. Generating executable goal-scenarios from natural language [25] also seems like a promising direction.

The extension we offer here deals only with discrete action and state spaces. It would seem interesting to develop a generalization algorithm, also built upon a scenario-based language and exploiting its structure, but one that would allow continuous actions and states as well. In addition, integrating smart meta-parameter learning methods can help yield optimized results, without added programming effort; and using a reward shaping function, which, given a goal-scenario and its current state, estimates its progress toward accomplishing its goal, can further improve the learning rate [26].

Acknowledgments

We would like to greatly thank Assaf Marron and Gera Weiss for their ideas and comments. The research was supported by an Advanced Research Grant to DH from the European Research Council (ERC), and by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

References

[1] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Form. Methods Syst. Des.*, vol. 19, pp. 45–80, 2001.

[2] D. Harel, "Can Programming Be Liberated, Period?" *Computer*, vol. 41, pp. 28–37, 2008.

[3] D. Harel, A. Marron, and G. Weiss, "Programming Coordinated Behavior in Java," in *Proc. 24th European Conference on Object-Oriented Programming*.

[4] ——, "Behavioral Programming," *Comm. Assoc. Comput. Mach.*, to appear.

[5] D. Harel and R. Marelly, "Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach," *Software and System Modeling (SoSyM)*, vol. 2, pp. 82–107, 2003.

[6] D. Harel, S. Maoz, S. Szekely, and D. Barkan, "PlayGo: Towards a Comprehensive Tool for Scenario Based Programming," in *Proc. of the IEEE/ACM int. conf. on Automated software engineering*, ser. ASE, 2010, pp. 359–360.

[7] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[8] N. Sprague and D. Ballard, "Multiple-Goal Reinforcement Learning with Modular Sarsa(0)," in *Proc. of the 18th int. joint conf. on Artificial intelligence*, 2003, pp. 1445–1447.

[9] S. J. Russell and A. Zimdars, "Q-Decomposition for Reinforcement Learning Agents," in *Proc. of the 20th Int. Conf. on Machine Learning*, ser. ICML, 2003, pp. 656–663.

[10] S. Singh, T. Jaakkola, M. L. Littman, and C. S. Ari, "Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms," in *Machine Learning*, 1998, pp. 287–308.

[11] J. Karlsson, "Learning to solve multiple goals," Ph.D. dissertation, University of Rochester, 1997.

[12] C. Simpkins, S. Bhat, C. Isbell, Jr., and M. Mateas, "Towards Adaptive Programming: Integrating Reinforcement Learning into a Programming Language," *SIGPLAN Not.*, vol. 43, pp. 603–614, 2008.

[13] S. Bhat, C. L. Isbell, and M. Mateas, "On the Difficulty of Modular Reinforcement Learning for Real-World Partial Programming," in *Proc. of the 21st National Conf. on Artificial intelligence - Volume 1*, 2006, pp. 318–323.

[14] J. Loch and S. Singh, "Using Eligibility Traces to Find the Best Memoryless Policy in Partially Observable Markov Decision Processes," in *Proc. of the 15th Int. Conf. on Machine Learning*, ser. ICML, 1998, pp. 323–331.

[15] N. Eitan, D. Harel, M. Gordon, A. Marron, and G. Weiss, "On Visualization and Comprehension of Scenario-Based Programs," in *Proc. of the 19th IEEE Int. Conf. on Program Comprehension*, ser. ICPC, 2011.

[16] BPJ Visualization site. [Online]. Available: http://www.cs.bgu.ac.il/~geraw/SupWebSite/

[17] JUNG. [Online]. Available: http://jung.sourceforge.net

[18] BPJ. [Online]. Available: http://www.cs.bgu.ac.il/~geraw/

[19] I. Ghory, "Reinforcement Learning in Board Games ," Department of Computer Science, University of Bristol, Tech. Rep., 2004.

[20] R. A. Brooks, "Elephants don't play chess," *Robotics and Autonomous Systems*, vol. 6, pp. 3–15, 1990.

[21] J. J. Bryson, "Cross-Paradigm Analysis of Autonomous Agent Architecture," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 12, no. 2, pp. 165–190, 2000.

[22] P. Maes, "Situated Agents Can Have Goals," *Robot. Auton. Syst.*, vol. 6, pp. 49–70, 1990.

[23] M. Mateas and A. Stern, "A Behavior Language For Story-Based Believable Agents," *Intelligent Systems, IEEE*, vol. 17, no. 4, pp. 39 – 47, 2002.

[24] D. Harel, H. Kugler, R. Marelly, and A. Pnueli, "Smart Play-out of Behavioral Requirements," in *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD, 2002, pp. 378–398.

[25] M. Gordon and D. Harel, "Generating Executable Scenarios from Natural Language," in *Proc. of the 10th Int. Conf. on Computational Linguistics and Intelligent Text Processing*, ser. CICLing, 2009, pp. 456–467.

[26] A. Y. Ng, D. Harada, and S. Russell, "Policy Invariance under Reward Transformations: Theory and Application to Reward Shaping," in *Proc. of the 16th Int. Conf. on Machine Learning*, 1999, pp. 278–287.